Use of Dependency Information for Memory Optimizations in Distributed Streaming Applications

Nissim Harel*, Hasnain A. Mandviwala*, Umakishore Ramachandran* and Kath Knobe†
*College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332
Email: {nissim, mandvi, rama}@cc.gatech.edu
†Intel Corporation, Email: kath.knobe@intel.com

Abstract—In this paper we explore the potential of using application data dependency information to reduce the average memory consumption in distributed streaming applications. By analyzing data dependencies during the application runtime, we can infer which data items are not going to influence the application’s output. This information is then incorporated into the garbage collector, extending the garbage identification problem to include not only data items that are not reachable, but also those data items that are not fully processed and dropped. We present three garbage collection algorithms. Each of the algorithms uses different data dependency information. We implement the algorithms and compare their performance for a color tracker application. Our results show that these algorithms not only succeed in substantially reducing the average memory usage but also improve the overall performance of the application. The results also indicate that the garbage identification algorithms achieve a low memory footprint perform their garbage identification decisions locally; however, they base these decisions on best-effort global information. The results also indicate that the garbage identification algorithms perform best when they base their decisions on best-effort global information obtained from other components of the distributed application.

I. INTRODUCTION

The physical and economic feasibility of capturing and processing a large number of data streams from different sources in real-time, make it possible to develop and deploy a new class of applications, called streaming applications. Broadly speaking, streaming applications are organized as a series (or a pipeline) of tasks processing streams of data, e.g., starting with sequences of camera images, extracting higher level “features” and “events” at each stage, and eventually responding with outputs. The applications tend to be distributed and involve the processing of large sets of different types of streaming sources of information at near real-time. These applications tend to be distributed and involve processing large sets of different types of streaming inputs at near real-time. The requirement to handle large quantities of data introduces a major challenge to the overall efficiency of streaming applications and makes effective memory and buffer management vital for a successful deployment of these applications.

The amount of computing power we currently have still allows us to process only a fraction of all the data captured by the application. Nevertheless, in many cases it is enough to get us sufficiently close to the desired result (i.e., what we would have achieved had we processed all the data). The reason lies in the fact that streaming applications try to attach a meaning to the information they acquire. The goal is not to fully process all the data captured, but rather extract a specific meaning from the data. The data items that are not fully processed are dropped at different stages of the computation and do not have influence on the application’s outcome. The resources (computation, bandwidth, and memory) that are allocated to process these items can, therefore, be considered as wasted and should be minimized.

In this paper, we exploit the unique characteristics of streaming applications [1], and explore the potential of using inter-stream data dependency information to identify and reclaim wasted resources as early as possible. In a broader context, we propose extending the definition of the garbage collection problem in streaming applications to include not only data items that are not “reachable” by the application’s threads, but also data items that have no effect on the final outcome of the application. Each one of the algorithms we propose applies a different method to identify items that can be considered as garbage. The algorithms analyze the allocated data items and use data dependency information to infer those items that would not be requested by any of the application threads. These items are denoted as garbage and reclaimed.

The proposed memory optimization algorithms are evaluated using the Stampede runtime system [2], which serves as a test bed for this study. Stampede is a runtime system that supports the development and execution of streaming applications by handling communication, synchronization, and buffer management, in-turn directing the application writer’s attention away from these arduous and repeated tasks. The programming model of Stampede is simple and intuitive. A Stampede program consists of a dynamic collection of threads communicating timestamped data items through channels. Threads can be created to run anywhere in the cluster. Channels can be created anywhere in the cluster and have cluster-wide unique names. Threads can connect to these channels to perform input/output via get/put operations. The facility through which channels and threads communicate is called a connection. A timestamp value is used as a name for a data item that a thread puts into or gets from a channel. The runtime system of Stampede takes care of the synchronization and communication inherent in these operations, as well as managing the storage for items put into or gotten from the channels.

The remainder of the paper is organized as follows. In
Section II we present three algorithms for optimizing memory usage in streaming applications. In Section III we discuss the implementation of the algorithms proposed in Section II. Performance results are provided in Section IV. Related work is provided in Section V, and we conclude with Section VI.

II. PROPOSED ALGORITHMS

In this section we present three algorithms for optimizing memory consumption in distributed streaming applications. Each algorithm explores the potential of using a different set of application data dependency information to identify and reclaim data items that the application is not going to use.

Dead timestamps based Garbage Collector (DGC): The DGC algorithm models a streaming application as a directed graph where nodes in this graph correspond to the application’s threads and channels, while edges correspond to input and output connections. During the application execution, the DGC algorithm analyzes local conditions at threads and channels, in conjunction with application knowledge, to generate a local timestamp guarantee on each node. The guarantee is a time marker such that all items with timestamps below its value are irrelevant and considered dead, while all timestamps equal or above its value are relevant and considered live. Using this local guarantee, the runtime system can determine whether a timestamped item can be labeled as garbage. This information is then propagated to the rest of the application graph nodes and is incorporated into their localized decision process of identifying items for garbage collection. As a result, the decision of whether an item should be reclaimed or not is performed locally, but it is based on global information obtained from all the nodes. This algorithm is further discussed in [3].

Keep Latest n if Unseen (KLnU): The KLnU algorithm takes advantage of a particular characteristic of streaming applications; namely, that in many cases downstream computations are interested only in the latest items produced by an upstream computation. In addition, upstream computations tend to be lighter than downstream computations. This trait of streaming applications result in a large number of items becoming irrelevant. The proposed algorithm is to associate an attribute with a channel (i.e., the memory buffer) that allows it to discard all but the latest items. When a producer puts a new item, the channel may immediately denote any item with an earlier timestamp as garbage if that item has not been gotten up until this point on any one of the channel connections. We refer to this set of earlier timestamps identified as garbage items as a dead set. We will make use of these sets in the algorithm presented later in this section.

Since a Stampede application depends on timestamp causality, this optimization will not allow an item with an earlier timestamp to be garbage collected even if one connection has gotten that item from this channel. We generalize this attribute and call it Keep Latest n Unseen (KLnU), to signify that a channel retains only the last $n$ items. The value of $n$ is specified at channel creation time and different channels may have a different value for $n$. Although, this attribute gives the channel a local control to garbage collect items that are deemed irrelevant, it can only be implemented as an addition to whatever system-wide garbage collection mechanism (e.g., DGC) is already in place.

Figure 1 illustrates an example of a channel operating under the KLnU algorithm, where $n=1$. The producer thread has just produced an item with timestamp 12 while items with timestamps 10 and 11 are already present in the channel. The item with timestamp 10 has been gotten by a consumer, while the item with timestamp 11 has not been gotten by any consumer so far. In other words, the item with timestamp 11 is “unseen” to all of the consumer threads connected to this channel. Thus, upon a put operation of an item with timestamp 12, the “unseen” item (11) can be added to the dead set of this channel and garbage collected while the “seen” items (in this case, the item with timestamp 10) is retained in the channel.

Propagating Dead Sets (PDS): KLnU is just one way to incorporate application knowledge to help in identifying garbage items in a channel more efficiently and generate local dead sets. One may think of other attributes and policies that may fit other data-dependency scenarios and that may generate dead sets within a channel in different ways. However, under each of these possible algorithms, the dead set information stays within a channel and does not propagate to other threads and channels in the application. The PDS algorithm propagates local dead-set information from one node to other parts of the application similar to the way the DGC algorithm propagates guarantees. Figure 2 shows the state of a node in a given task graph. This node updates its dead set by incorporating local data dependency information via mechanisms such as KLnU. It also includes propagated dead-set information received from all the output connections (out-edges) due to backward propagation. Similarly, it adds dead-set information to the local set from all the input connections (in-edges) due to forward propagation. The dead-set for a node is computed as the union of the local dead-set and the intersection of the dead set information of all the in and out-edges (or connections) incident at that node.
III. IMPLEMENTATION

All GC algorithms are implemented in Stampede[2], a runtime system that supports the development and execution of streaming applications. The Stampede runtime system serves as a test bed for this study.

Both KLnU and PDS do not replace the garbage collector, but instead optimize the performance of the runtime system’s garbage collector. KLnU is capable of identifying some data items as garbage earlier than the existing garbage collector does. PDS is able to propagate information about these data items to other nodes in the application faster than any baseline garbage collector in the runtime system. They are implemented on top of the DGC algorithm so that the PDS implementation can use DGC’s guarantees propagation mechanism for the propagation of dead set information. In addition, a prior study [3] that compares GC algorithms in Stampede indicates that DGC uses up to 40% less memory than other GC algorithms in Stampede. The details of the DGC implementation are discussed in [3].

KLnU optimization associates an attribute with a channel that allows it to discard all but the latest items. In Stampede, put operations to a channel have the potential of activating the KLnU optimization, and discard those items that were previously transmitted to the channel, but have not yet been gotten by any one of the relevant threads. Therefore, KLnU can be implemented by adding a function within the Stampede put operation that checks whether the channel holds items that were not gotten by any connection. These items are tagged as garbage and discarded. The Keep Latest $n$ if Unseen (KLnU) optimization keeps only the last $n$ unseen items in a channel, where $n$ is a parameter that the application writer may set as required by the application. This parameter provides the application with an n-sized window “into the past”. In the case of the color tracker application, however, the application is interested only in the latest item, and thus $n$ can be set to 1.

PDS uses the dead set information generated by the KLnU algorithm and propagates it to neighboring nodes. It maintains a list of dead items and every time a new item is identified as garbage, it is added to this list of dead items. This list is then propagated upon Stampede put operations to channels earlier in the application pipeline. As items are added to the dead set, PDS maintains the set to prevent it from growing indefinitely by removing from the set those items the underlying garbage collector has already identified as garbage. It uses the low water mark of the underlying garbage collector to trim items that are already considered as garbage by the garbage collector.

In our case, where DGC serves as the underlying garbage collector, items with timestamps that are below the node guarantee can be removed from the dead set, as the garbage collector has already labeled them as garbage to be reclaimed.

IV. PERFORMANCE OF GC ALGORITHMS

A. Setup and Metrics

We use a real-time color-based people tracker application [4] for this study. Given a color histogram model of an object, this application locates the object in a scene if present. The application task graph and its connection dependencies are provided in [2]. A digitizer produces a new image of the scene every 30 milliseconds. In this experiment, the digitizer reads a pre-recorded set of images from a file to have a fair comparison among the three GC algorithms. A digitized image is then passed to both a motion detection thread and a histogram thread. The motion detection thread subtracts the current image from the former image to detect changes in the scene. The histogram thread receives input from both the digitizer and the motion detection threads and generates a color histogram of the image received. The target detection thread uses input from the digitizer, the motion detection, and the histogram threads to locate a pre-determined target in the image. This thread is the slowest stage in the application pipeline and cannot keep up with the digitizer’s rate of frame production. Thus, not every image the digitizer produces will propagate through the entire pipeline, and successfully reach the end. Every pipeline stage gets the latest available timestamp from their respective input channel connections. A real deployment of this application may use multiple target detection threads (one per target being tracked). For our experiments, we use two target detection threads, each searching for a different object in the same frame using a color-histogram model unique to the object. With this workload, the average message sizes delivered to the digitizer, motion mask, histogram, and target detection channels are 738KB, 246KB, 981KB, and 67B, respectively.

We use two configurations of the color tracker application: (1) 1-node configuration - All threads execute on one physical node within a single address space. The channels are also mapped to the same physical node. No network-related communication occurs in this configuration; and, (2) 5-node configuration - Each one of the five threads (and their corresponding output channels) is mapped to a distinct node in a cluster. As a result, the application is distributed over five physical nodes of the cluster. This configuration maximizes the network-related communication of the application. In addition, threads locally cache a copy of the item they access because
they cannot access directly items that are allocated in channels that are mapped to a different node. Thus, this configuration requires also additional storage compared to the 1-node configuration. The experiments are performed on a cluster of SMPs interconnected by Gigabit Ethernet. Each node is an 8-way SMP, comprising of 550 MHz Intel Pentium III Xeon processors with 4GB of physical memory. The Linux kernel used is Redhat 2.6.9, along with a reliable UDP messaging library called CLF (Cluster Language Framework) [5].

We use the following metrics to evaluate the KLnU and the PDS optimizations:

**Average Memory Usage** - This metric quantifies the aggregate memory used by the application in all the channels averaged over the execution time of the application. It is a measure of the average memory pressure exerted by the application during the course of the execution. The more aggressive GC algorithm the runtime system uses, data items are reclaimed faster and the memory pressure exerted on the system is reduced.

**Total Channel Occupancy Time** - This metric quantifies the total time spent by all items in channels before they are garbage collected. In addition, we distinguish between used occupancy time, time spent on “successful” items, i.e., those items that make it through the application pipeline and affect the output, and wasted occupancy time, time spent on items that do not reach the end of the application pipeline and therefore have no influence on the end result.

**Average Pipeline Latency** - This metric quantifies the average latency experienced by items that make their way through the entire application pipeline. Latency is measured from the time an item enters the first pipeline stage, until it leaves the last pipeline stage.

**Average Throughput** - This metric measures the average number of items that reach the end of the pipeline, and thus affects the final outcome. The larger the throughput the more items the application can process (on average) in a given time.

### B. Results

The implementation results presented in Table I show that both KLnU and PDS optimizations reduce the amount of memory consumed by the application. When KLnU is implemented on top of DGC there is a 4% and a 14% reduction in average memory usage for the color tracker application in the 5-node and the 1-node configurations, respectively. When PDS is implemented on top of DGC there is a 31% and a 60% reduction in average memory usage for the color tracker application in the 5-node and the 1-node configurations, respectively.

The KLnU and PDS optimizations not only reduce the average memory usage, but also improve the application performance in terms of latency and throughput. As the runtime system reduces its overhead (maintaining items that will not reach the end of the application pipeline) and directs its resources towards productive work (that is, towards items that will successfully reach the end of the application pipeline), data items are able to reach the end of the pipeline faster and more frequently. This performance improvement offsets the additional overhead that KLnU and PDS introduce. Under KLnU, the garbage identification decision is made by the channel and is based on information readily available to the channel. Thus, the overhead associated with KLnU is relatively small. PDS, on the other hand, has higher overhead because it propagates additional information to neighboring nodes. This overhead increases as communication costs increase. As a result, the PDS overhead is even greater in the 5-node configuration. In summary, the overall performance of the application is determined by a balance between the benefits of the reduced overhead by maintaining a smaller number of items, and the increased overhead of the implemented KLnU and PDS optimizations.

Throughput under both optimizations increases rather than decreases. Table I shows that KLnU increases the application throughput from 4.52[fps] to 4.74[fps] (or by 5%) for the 1-node configuration, and from 4.12[fps] to 4.27[fps] (or by 4%) for the 5-node configurations. PDS increases the application throughput from 4.52[fps] to 4.77[fps] (or by 6%) for the 1-node configuration, and from 4.12[fps] to 4.23[fps] (or by 3%) for the 5-node configuration. Similarly, latency does not increase but rather decreases. Table I shows that under KLnU the latency decreases from 389[ms] to 374[ms] (or by 4%) for the 1-node configuration, and from 628[ms] to 358[ms] (or by 8%) for the 5-node configuration. Under PDS the latency decreases from 389[ms] to 362[ms] (or by 7%) for the 1-node configuration; however, the latency increases slightly from 628[ms] to 634[ms] (or by less than 1%) for the 5-node configuration. This is the result of the relatively large overhead associated with PDS under the 5-node configuration.

Tables II and III present the total occupancy time (that is, the total time items spent in the channels) for the 5-node and the 1-node configurations, respectively. They break the total occupancy time into wasted and used categories. Wasted occupancy time is the aggregate time items that do not reach the end of the pipeline spend in channels. This occupancy time does not contribute to the application end result, and thus, can be viewed as wasted. Used occupancy time, on the other hand, is the aggregate time successful items spend in channels. These items successfully reach the end of the pipeline, and affect the application output. Therefore, it is constructive to direct resources towards managing these data items.

<table>
<thead>
<tr>
<th>Setup</th>
<th>Config 1: 1 node</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory Usage (MB)</td>
<td>Latency (ms)</td>
<td>Throughput (fps)</td>
<td></td>
</tr>
<tr>
<td>DGC</td>
<td>16.36</td>
<td>389</td>
<td>4.52</td>
<td></td>
</tr>
<tr>
<td>DGC+KLnU</td>
<td>14.06</td>
<td>374</td>
<td>4.74</td>
<td></td>
</tr>
<tr>
<td>DGC+PDS</td>
<td>6.50</td>
<td>362</td>
<td>4.77</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Setup</th>
<th>Config 2: 5 nodes</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory Usage (MB)</td>
<td>Latency (ms)</td>
<td>Throughput (fps)</td>
<td></td>
</tr>
<tr>
<td>DGC</td>
<td>25.03</td>
<td>628</td>
<td>4.12</td>
<td></td>
</tr>
<tr>
<td>DGC+KLnU</td>
<td>24.06</td>
<td>580</td>
<td>4.27</td>
<td></td>
</tr>
<tr>
<td>DGC+PDS</td>
<td>17.29</td>
<td>634</td>
<td>4.23</td>
<td></td>
</tr>
</tbody>
</table>

### Tables II and III

AVERAGE MEMORY USAGE, LATENCY, AND THROUGHPUT.
TABLE II
CHANNEL OCCUPANCY TIME.

<table>
<thead>
<tr>
<th>Setup</th>
<th>Config 1: 1 node</th>
<th>Config 2: 5 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wasted Occupancy Time (sec)</td>
<td>Used Occupancy Time (sec)</td>
</tr>
<tr>
<td>DGC</td>
<td>3.784</td>
<td>991</td>
</tr>
<tr>
<td>DGC+KLaU</td>
<td>2.386</td>
<td>995</td>
</tr>
<tr>
<td>DGC+PDS</td>
<td>604</td>
<td>949</td>
</tr>
<tr>
<td>DGC</td>
<td>2.913</td>
<td>3,637</td>
</tr>
<tr>
<td>DGC+KLaU</td>
<td>2.685</td>
<td>3,296</td>
</tr>
<tr>
<td>DGC+PDS</td>
<td>875</td>
<td>3,633</td>
</tr>
</tbody>
</table>

V. RELATED WORK

The traditional GC problem (on which there is a large body of literature, for example, [6], [7]) concerns reclaiming storage for heap-allocated objects (data structures) when they are no longer “reachable” from the computation. The “name” of an object is a heap address, i.e., a pointer, and GC concerns a transitive computation that locates all objects that are reachable starting with names in a symbol table. In most safe GC languages, there are no computational operations to generate new names (such as pointer arithmetic) other than the allocation of a new object. Stampede’s GC poses an orthogonal problem. The “name” of an object in a channel is its timestamp, i.e., the timestamp is an index or a tag. Timestamps are simply integers, and threads can compute new timestamps. GC of timestamps is concerned with determining when a timestamped item will not be used any more (regardless of whether it is reachable) and thus, storage associated with all or some items that are tagged with this timestamp can be reclaimed.

The problem of determining the interest set for timestamp values in the Stampede programming model has similarity to the garbage collection problem in Parallel Discrete Event Simulation (PDES) systems [8], yet it is less restrictive. Unlike the Stampede programming model, PDES systems require that repeated executions of an application program using the same input data and parameters produce the same results [9]. To ensure this property, every timestamp must appear in order by the PDES system. A number of synchronization algorithms have been proposed in the PDES literature to preserve this property. First attempts to perform GC were based on conservative assumptions. Algorithms such as Chandy-Misra-Bryant (CMB) [10], [11], for example, process the timestamps strictly in order, exchanging null messages to avoid potential deadlocks. Decisions are made locally, and there is no reliance on any global mechanism or control. Optimistic algorithms, such as Time Warp [12], assume that processing a timestamp out of order by a node is safe. However, if this assumption proves false, then the node rolls back to the state prior to processing the timestamp. To support such a roll back, the system has to keep around state, which is reclaimed based on calculation of a Global Virtual Time (GVT). The tradeoff between the conservative (CMB) and optimistic (Time Warp) algorithms is space versus time.
While the former is frugal with space at the expense of time, the latter does the opposite. On the other hand, the Stampede programming model does not require in-order execution of timestamps, nor does it require that every timestamp be processed. Consequently, it does not have to support rollbacks. If nothing is known about the application task graph, then similar to PDES, there is a necessity in the Stampede programming model to compute GVT to enable garbage collection. The less restrictive nature of this programming model allows conception of different types of garbage collection algorithms. These algorithms use application-level knowledge enabling garbage collection based entirely on local events with no reliance on any global mechanism. Like CMB, they are frugal in space; however, application-level knowledge enables them to reduce latency while reducing space requirements. Gaining performance in both fronts is achieved by breaking the boundary between the application and the run-time system. Whether it is possible to achieve similar gains in PDES systems is an interesting problem worthy of investigation.

VI. CONCLUSIONS

In this paper we demonstrate the effectiveness of using application data dependency information to reduce the memory footprint of distributed streaming applications. We also show how the decision can be realized as a natural extension of the traditional garbage collection model. Of the three algorithms we present, KLnU makes local decisions based on local information, and thus, has no communication overheads. On the other hand, the memory reduction achieved is relatively small (up to 14% compared to the baseline DGC), as dependency information is not shared with other nodes. Conversely, both DGC and PDS algorithms propagate local data dependency information, and as a result, can base their local decisions on global information. The results demonstrate the value of this combination. Although DGC does not harness the full garbage identification potential prevalent in the system, it still manages to achieve up to 40% reduction in memory consumption compared to other garbage collection algorithms in Stampede. PDS improves the DGC performance by propagating local dependency information found by the KLnU algorithm in addition to the local information captured by the DGC algorithm. PDS succeeds in achieving up to 60% reduction in average memory usage compared to baseline DGC. An important finding is that the overheads associated with analyzing application dependencies and propagating this information to the rest of the application’s nodes are compensated by the reduction in the overheads associated with maintaining data items that the application no longer needs. In fact, not only that this substantial reduction in memory consumption does not hurt the application’s performance, it actually improves it.

REFERENCES


